C13

Function Index

```
1   /*
2   ** Copyright 1996,1997 BMC Corporation
3   **
4   **
5   ** EDMDispatch.c
6   **
7   **
8   ** Mission Statement: this is the main service file for the dispatch
9   **                    daemon. this file contains the callbacks from the main
10  **                    which prepares the daemon to go off and service
11  **                    RPC's.
12  **
13  **
14  ** Primary Data Acted On:
15  **
16  **        None.
17  **
18  ** Compile-Time Options:
19  **
20  **        None.
21  **
22  ** Basic idea here: Module for UNIX specific daemon initialization
23  **
24  ** The following provides an RCS id in the binary that can be located
25  ** with the what(1) utility. The intent is to keep this short.
26  */
27  static char     RCS_id [] = "@(#)$RCSfile: EDMDispatch.c,v $ \
28                              "$Revision: 1.23 $ \
29                              "$Date: 1997/02/06 20:49:15 $" ;
30  #endif
31  /* #define _POSIX_SOURCE       unable to compile with this define set */
32  /* #define _XOPEN_SOURCE       unable to compile with this define set */
33
34  #include <esl/c_portable.h>
35  #include <esl/ep_xopen.h>
36  #include <esl/inout.h>
37
38  #include <stdarg.h>
39  #include <string.h>
40  #include <syslog.h>
41  #include <thread.h>
42  #include <thread.h>
43  #include <sys/resource.h>
44
45  #include <EDMmain.h>
46  #include <EDMD.ccw.h>
47  #include <EDMDispatchlog.h>
48  #include <EDMDispatchbackground.h>
49  #include <util/esl_daemon.h>
50  #include <csc/cscom.h>
51  #include <EDMDCI_rsrsc.h>
53  #include <restore/csc_EDMDispatch.h>
54
55  /*
56  ** Need to define _XOPEN_SOURCE for signal function definitions
57  ** and certain signal structure definitions.
59  */
63  #define _XOPEN_SOURCE
```

```
65  #include <signal.h>
67  #undef _XOPEN_SOURCE
69  static rpc_if_handle_t if_spec;
71  static int      g_debug = FALSE;   /* Variable which will disable forking */
73  static char     **commandlineargs ; /* Pointer to command line args */
75  /* *********************************************************** */
76  /* *                                                         * */
77  /* *  Routine: IsDebugOn                                     * */
78  /* *                                                         * */
79  /* *  Inputs: None                                           * */
80  /* *                                                         * */
81  /* *  Outputs: None                                          * */
82  /* *                                                         * */
83  /* *  Return Codes:                                          * */
84  /* *                                                         * */
85  /* *                 TRUE if debug is on.                    * */
86  /* *                                                         * */
87  /* *  Purpose:                                               * */
88  /* *     This routine can be used to tell other subsystems   * */
89  /* *     whether debugging is available.                     * */
90  /* *                                                         * */
91  /* *  Intended caller:   internal only.                      * */
92  /* *                                                         * */
93  /* *********************************************************** */
94  boolean_t
95  IsDebugOn()
96  {
97      return g_debug;
    }
```

```
 99   /**********************************************************
100    **
101    ** Routine: kill_handler
102    **
103    ** Inputs: int signal - the signal which was received
104    **
105    ** Outputs: Will log messages telling what action is being taken.
106    **
107    ** Return Codes:
108    **
109    **      exits with the number of the signal received
110    **
111    ** Purpose:  This routine handles specific signals i.e. SIGINT,
112    **           SIGQUIT,
113    **           SIGTERM. Each results in a log entry and an exit.
114    **
115    ** Intended caller: internal only.
116    **
117    **
118    **********************************************************/
118   static void kill_handler( IN int signal )
119   {
120       int       status;
121       time_t    current_time;
122       char      *ctimebuf;
123       char      *ebuf;
124
124       /* If main exits, it calls this routine with signal 0 */
128
129       /* Unregister the interface */
130       (void) csc_unregister_server_interface(&if_spec, &status);
131       if ( status != error_status_ok )
132       {
133           /* If the unregister fails, report the problem, but continue */
134           ebuf = (char *) csc_get_error( status );
135
136           EDMDispatch_logent(
137               __FILE__, __LINE__, LOG_ERR, MESSAGE_NO_LOGIN, 0,
138               "CSC_SERVER_LOGIN failed: <%d> %s",
139               status, (ebuf ? ebuf : "Unknown error") );
140       }
141
142       /* Get the current time */
143       time(&current_time);
144
145       ctimebuf = ctime(&current_time);
146
147       /* Overlay newline with null - buf should always be 26 bytes long */
148       ctimebuf[ strlen(ctimebuf) - 1 ] = 0;
149
150       EDMDispatch_logent(
151           __FILE__, __LINE__, LOG_INFO, MESSAGE_SHUTDOWN, 0,
152           "Shutting down at %s due to signal %d", ctimebuf,
153           signal);
155
157       /* Remove our lock file. */
           (void) EdDestroyPidFile(PIDPATH);

           exit(signal);

       } /* End of kill_handler() */
```

```
159   /**********************************************************
160    **
161    ** Function Name:
162    **     display_usage
163    **
164    ** Simply displays the usage
165    **
166    ** Call Arguments:
167    **     Program name
168    **
169    ** Error Outputs and Side Effects:
170    **     Prints usage.
171    **
172    ** Special Considerations:
173    **     None.
174    **
175    **/
176   static void
177   display_usage (IN char *progname)
178   {
179       /* Print out usage stmt. */
180
181       fprintf (stderr, "Usage: %s [-d]\t", progname);
182       fprintf
184       fprintf (stderr, "-d keep the daemon from forking so debugging is easier\n");
186   } /* end display_usage () */
```

```
187
188        /*
189        ***********************************************************
190        ** Routine: daemon_catch_interrupts
191        **
192        ** Inputs:     None
193        **
194        ** Outputs:    None
195        **
196        ** Return Codes:  None
197        **
198        ** Purpose:     Sets up signals for service. On NT we will have to
199                       consider what OS constructs to replace signals with.
                          In this case we are catching SIGTERM, SIGINT, and
                          SIGQUIT and ignoring anything else.
202        **
203        ** Intended caller: internal only.
204        **
205        ***********************************************************
206        */
207        void daemon_catch_interrupts()
208        {
209        struct sigaction sactions;                 /* Signal actions */
210
211             /* Set an empty list so we can set signals we want to handle */
                (void) sigemptyset( &sactions.sa_mask );
213
216             /* Add signals that we want to handle */
217
218             (void) sigaddset( &sactions.sa_mask, SIGTERM );
219             (void) sigaddset( &sactions.sa_mask, SIGINT );
220             (void) sigaddset( &sactions.sa_mask, SIGQUIT );
221
222             /* Setup the signal handler */
223             sactions.sa_handler = kill_handler;
224
225             /* Assign handler to each signal we are interested in. */
226
228             (void) sigaction( SIGTERM, &sactions, NULL );
229             (void) sigaction( SIGINT, &sactions, NULL );
230             (void) sigaction( SIGQUIT, &sactions, NULL );
231
232             /* Setup mask so we can specify what signals we will ignore. */
233             (void) sigfillset( &sactions.sa_mask );
234
235             /* We want to ignore everything except those we have set up
236              * above so remove those from the list.
237              */
238             (void) sigdelset( &sactions.sa_mask, SIGTERM );
240             (void) sigdelset( &sactions.sa_mask, SIGINT );
241             (void) sigdelset( &sactions.sa_mask, SIGQUIT );
242
243
244
245
246
247
248
```

```
249             * Set the mask. Since no other threads have been started
250             * all threads will get this mask.
251             */
252             (void) thr_sigsetmask( STG_SETMASK, &sactions.sa_mask, NULL, );
253        }
```

```
256  /*********************************************************
257  **
258  ** Routine: daemon_check_proper_ID
259  **
260  ** Inputs:
261  **      None
262  **
263  ** Outputs:
264  **      None
265  **
266  ** Return Codes:
267  **      exits with an error when the user is not root
268  **
269  ** Purpose:
270  **      Checks user's ID and determines if the user is allowed
271  **      to execute service. If there are no constraints then this
272  **      function may be blank.
273  **
274  ** Intended caller: internal only.
275  **
276  **
277  *********************************************************/
278  void daemon_check_proper_ID()
279  {
280      /*
281      ** Check for root
282      */
283      if (geteuid() != E_ROOTUID)
284      { (void) EDMDispatch_logmt(
285                  __FILE__, __LINE__, LOG_ERR, DAEMON_NOTSUPERUSER, 0,
286                  "Must be run as superuser, uid was %d",
287                  geteuid());
288          exit(1);
289      }
     }
```

```
291  /*********************************************************
292  **
293  ** Routine: parse_commandline
294  **
295  ** Inputs:
296  **      argc, argv (command line arguments)
297  **
298  ** Outputs:
299  **      None
300  **
301  ** Return Codes:
302  **      exits with an error when the user types a bad argument
303  **
304  ** Purpose:
305  **      Parse command line arguments and sets flags. If there
306  **      are no flags to be set then this function may be empty.
307  **
308  ** Intended caller: internal only.
309  **
310  **
311  *********************************************************/
312  void parse_commandline(int argc, char *argv[])
313  {
314      int     opt;
315      commandlineargs = argv;
316      while ((opt = getopt(argc, argv, "dD")) != EOF)  /* Process options */
317      { switch(opt)
318          { case 'd':
319            case 'D':
320                  g_debug = TRUE;
321                  break;
322            default:
323                  (void) display_usage( argv[0] );
324                  exit(1);
325          }
326      }
327  }
```

```
332
333  /********************************************************************
334  **
335  ** Routine: daemon_initialize_logging
336  **
337  ** Inputs:     None
338  **
339  ** Outputs:    None
340  **
341  ** Return Codes: None
342  **
343  ** Purpose:    Do whatever it takes to initialize logging.
344  **             In the near
345  **             future this may involve doing something with catalogs or
346  **             calling higher level logging functions which encapsulate
347  **             these things.
348  **
349  ** Intended caller: internal only.
350  **
351  *********************************************************************
352  */
353  void
354  daemon_initialize_logging ()
355  {
356      /* Pass in argv[0], the program name */
357      (void) esl_log_init(commandlinearqs[0]);
358  }
```

```
360
361  /********************************************************************
362  **
363  ** Routine: daemon_become_daemon
364  **
365  ** Inputs:     None
366  **
367  ** Outputs:    None
368  **
369  ** Return Codes: None
370  **
371  ** Purpose:    This function is for doing the forking etc. under UNIX.
372  **             It is unknown what will be necessary under NT.
373  **
374  ** Intended caller: internal only.
375  **
376  *********************************************************************
377  */
378  void
379  daemon_become_daemon ()
380  {
381      char *ptr;
382      int ret = 0;
383
384      /*
385      ** Strip the path from the program name so we can use it
386      ** elsewhere.
387      */
388      ptr = strrchr(commandlinearqs[0], '/');
389      if (ptr != NULL)
390          ptr++;
391      else
392          ptr = commandlinearqs[0];
393
394      /* Change directory to a process specific core directory */
395      ret = esl_coredir_setup(ptr);
396      if (ret != 0)
397      {
398          (void) EDMDispatch_Logger(   __FILE__,   __LINE__, LOG_ERR,
399              MESSAGE_ERR_IN_ESL_COREDIR, 0,
400              "esl_coredir_setup failed <%d>",
401              errno);
402
403          exit(1);
404      }
405
406      /*
407      ** This is now esl functionality.  This code does everything necessary
408      ** to make this a "real" daemon by detaching from the terminal
409      ** changing the process group, closing stdout, stderr, stdin,
410      ** ...
411      */
412      if (g_debug == FALSE)
413      {
414          ret = esl_daemon_startup();
415          if (ret != 0)
416          {
417              fprintf(
418              stderr, "%s: Failed to initialize as daemon.\n",
                     commandlinearqs[0]);
```

```
419  3              }
420  2          }
421  1      }
421  1  }
422  1  }

           exit(1);
```

```
424      /**************************************************************
425      **
426      **  Routine:  rpc_init
427      **
428      **  Inputs:        None
429      **
430      **  Outputs:       None
431      **
432      **  Return Codes:
433      **      exits with an error code if initialization fails
434      **
435      **  Purpose:
436      **      This function is for doing RPC initialization.
437      **      For the most part it involves calling the rest routines.
438      **      This is pretty standard between UNIX and NT.
439      **
440      **  Intended caller:   internal only.
441      **
442      */
443
444  1  void rpc_init()
445  1  {
446  1      error_status_t      status;        /* error status (nbase.h) */
447  1      char    *ebuff;
448
449  1      /*
450  1      ** This is here because of HP which may or may not define timeval.
451  1      ** May be removed when esl_timeval is ported to clients
452  1      */
453  1  #ifdef _STRUCT_TIMEVAL
454  1      struct timeval  sleep_interval = {5,0}; /* 5 second sleep interval */
455  1  #else
456  1      struct timespec sleep_interval = {5,0}; /* 5 second sleep interval */
457  1  #endif
458
459  1      /* Setup the interface specification for RPC */
460  1      SERVER_IFSPEC(if_spec);
461
462  1      /*
463  1       * Login as SERVER_PRINCIPAL.  The context of the process
464  1       * will be set to this principal.
465  1       *
466  1       * This process will keep trying to login to DCE if the
467  1       * server is unavailable.   Note that under SUN RPC this is a no-op.
468  1       */
469  1      while (TRUE)
470  1      {
471  2          (void) csc_server_login(SERVER_PRINCIPAL,
472  2                  SERVER_KEYTAB, &status);
473
474  2          /* If we succeeded, then exit this loop. */
475  2          if ( status == error_status_ok )
476  2          {
477  3              break;
478  3          }
479  2          else  /* Print error message if appropriate. */
480  3          {
481  3              ebuff = (char *) csc_get_error( status );
```

```
483  3
484  3            (void) EDMDispatch_logent ( __FILE__, __LINE__, LOG_ERR,
485  3                                        MESSAGE_NO_LOGIN, 0,
486  3                                        "CSC_SERVER_LOGIN failed: <%d>
                                              %s",
487  3                                        status, (
488  2                              ebuff ? ebuff : "Unknown error"));
489  2            }
490  2
491  2            /* If the failure was due to unavailable client,
492  2             * pause and then try again.
493  3             */
494  3            if (status == sec_rgy_server_unavailable)
495  3            {
496  2                /*
497  2                 * uses sleep when SUNRPC; otherwise uses
498  3                 * pthread call to delay for the specified
499  3                 * time
500  2                 */
501  2                CSC_SLEEP(sleep_interval);
502  2                continue;
503  2            }
504  2
505  2            /* If we got here, we had a unexpected failure. */
506  1            (void) EDMDispatch_logent ( __FILE__, __LINE__, LOG_ERR,
507  1                                        MESSAGE_NO_LOGIN, 0,
508  2                                        "The service cannot log in as
                                              required");
509  1        }
510  1
511  1        exit(1);
512  1    }
513  1
514  1    /*
515  1    ** We need to initialize the authorization module before we do
516  1    ** a listen.
517  2    */
518  2    (void) csc_authorization_init(&status);
519  2
520  2    if ( status != error_status_ok )
521  2    {
522  2        ebuff = (char *) csc_get_error( status );
523  2
524  2        (void) EDMDispatch_logent ( __FILE__, __LINE__, LOG_ERR,
525  2                                    MESSAGE_NOAUTHORIZATION, 0,
526  1                                    "CSC_AUTHORIZATION_INIT failed: <%d> %s",
527  1                                    status, (
528  1                        ebuff ? ebuff : "Unknown error") );
529  1        exit(1);
530  1    }
531  1
532  2    (void) csc_register_server_interface(
533  2                                    &if_spec,
534  2                                    SERVER_ANNOTATION,
535  2                                    &status);
536  1
537  2    if ( status != error_status_ok )
538  2    {
539  2        ebuff = (char *) csc_get_error( status );
540  2
                (void) EDMDispatch_logent ( __FILE__, __LINE__, LOG_ERR,
                                            MESSAGE_CANNOTREGISTER, 0,
                                            "CSC_REGISTER_SERVER_INTERFACE failed:
                                            <%d> %s",
```

```
541  2                                    status, (
542  2                        ebuff ? ebuff : "Unknown error") );
543  1        exit(1);
            }
```

```
545
546  **
547  ** Routine:  rpc_run
548  **
549  ** Inputs:       None
550  **
551  ** Outputs:      None
552  **
553  ** Return Codes: None
554  **
555  ** Purpose:      This function is for running the RPC lister.
556  **               This is pretty standard between UNIX and NT.
557  **
558  ** Intended caller:  internal only.
559  **
560  ** *******************************************************************
561  */
562  void rpc_run()
563  {
564  error_status_t    status;          /* error status (nbase.h) */
565
566      char    *ebuff;
567
568      /* listen for RPC calls forever. */
569      (void) csc_server_listen(
570          rpc_c_listen_max_calls, &status );
571
572      ebuff = (char *) csc_get_error_status ( &status );
573
574      /* We don't expect to get here. */
575      (void) EDMDispatch_logent(
576          _FILE_, _LINE_, LOG_ERR, MESSAGE_SERVERLISTEN, 0,
577          "CSC_SERVER_LISTEN failed: <%b> %s",
578          status, ebuff ? ebuff : "Unknown error" );
     }
```

```
580  /* *******************************************************************
581  ** Routine:  daemon_specific_initialization
582  **
583  ** Inputs:       None
584  **
585  ** Outputs:      None
586  **
587  ** Return Codes: None
588  **
589  ** Purpose:      Do whatever makes this daemon special.  In some cases you
590  **               may want to start a thread or open a socket.
591  **               Do that here.
592  **
593  ** Intended caller:  internal only.
594  **
595  ** *******************************************************************
596  */
597  void
598  daemon_specific_initialization()
599  {
600  error_status_t    status;          /* error status (nbase.h) */
601      int              ret;
602      pthread_t        pthread_id;
603      pthread_t        pthread_id;          mantid;
604      time_t           current_time;        cleanid;
605      char            *ctimebuf;
606      struct rlimit rlp;
607
608
609      /* Create a file and lock it so we don't start multiple
610      ** daemons.  Exit if there is another copy of us running.
611      ** The createPidFile call already logs errors so just exit.
612      */
613      if  (IsIfcreatePidFile(PIDPATH))
614      {
615          exit(1);
616      }
617
618      /* Find out what time it is */
619      (void) time(&current_time);
620      ctimebuf = ctime(&current_time);
621
622      /* Overlay newline with null - buf should always be 26 bytes
623         long */
624      ctimebuf[ strlen(ctimebuf) - 1 ] = 0;
625
627      /* Log startup message */
628      (void) EDMDispatch_logent(
629          _FILE_, _LINE_, LOG_INFO, MESSAGE_STARTUP, 0,
630          "Restore service %s starting up",
631              commandlineargs[0],
632          );
633
634      /* set the open files limit (very large) */
635      rlp.rlim_max = FD_SETSIZE;
636      rlp.rlim_cur = FD_SETSIZE;
         setrlimit(RLIMIT_NOFILE, &rlp);
```

```c
638  1
640  1      getrlimit(RLIMIT_NOFILE, &rlp);
           (void) EDMDispatch_logent(
                __FILE__, __LINE__, LOG_INFO, MESSAGE_STARTUP, 0,
641  1                "Service allows %d open files",
641  1                rlp.rlim_max );

           /* Initialise service launcher */
641  1      ret = EDMDSvcInit();
643  1
643  1      if (ret != 0)
           {
661  1          (void) EDMDispatch_logent(
661  1              __FILE__, __LINE__, LOG_INFO, 0, 0,
648  2                  "Service Launcher failed returning - %d",
648  2                  ret);

649  2          exit(1);
           }

650  2
651  1      /*
653  1       * Start the other threads in the daemon. The main thread
654  1       * becomes the RPC thread. BANDataManage is the entry point
655  1       * for the RPC thread. BANDataCleanup is the
656  1       * for the data collection thread.
657  1       * entry point for the data expiration thread.
658  1       */
659  1      /*pthread_create(&amid, NULL, DispDaemon_ccr, NULL); */
660  1      pthread_create(&amid, NULL, DispDaemon_CCW, NULL);
661  1      pthread_create(&amid, NULL, DispDaemon_CCW, NULL);
662  1      pthread_create(&client1d, NULL, DispatchBackground, NULL);
663  1      rpc_init();
664  1      rpc_run();
           }
```

```c
666
667  /************************************************************
668   * Routine: daemon_cleanup
669   *
670   * Inputs:        None
671   *
672   * Outputs:       None
673   *
674   * Return Codes:  None
675   *
676   * Purpose:       Call function which will clean up daemon properly.
677   *
678   * Intended caller:   Internal only.
679   *
680   *
681   ************************************************************/
682
684  void
685  daemon_cleanup()
686  {
687      kill_handler( 0 );
688  }
```

```
1    /*
2    ** Copyright 1996,1997 BMC Corporation
3    */

6    /*
7    ** EDMDispatchService.c
8    **
9    ** Mission Statement: RPC entry points.
10   **
11   ** Primary Data Acted On:
12   **
13   ** Compile-Time Options:
14   **
15   ** Basic idea here:
16

18   #if !defined(lint)
19   static char    RCS_id [] = "@(#)$RCSfile: EDMDispatchService.c,v $ "
20                              "$Revision: 1.0 $ "
21                              "$Date: 1997/02/06 20:49:15 $" ;
22   #endif

24   #include <osl/c_portable.h>
25   #include <osl/inout.h>

27   #include <logging/logging.h>
28   #include <osc/oscomm.h>

30   #include <restore/osc_EDMDispatch.h>
31   #include <restore/dispatch_daemon.h>

33   #include <EDMDispatching.h>
34   #include <EDMDispatchSession.h>

36   /*
37   ** These are all the rpc entry points for the dispatch daemon.
38   ** The dispatch daemon is multi-threaded and it is the main thread
39   ** which handles all incoming RPC. ONC RPC is single threaded
40   ** which provides us some
41   ** safety in the way we handle our data and limits our exposure
42   ** to unexpected multithreading problems.
43   */
44   static void FreeSessionInfo(SessionInfo *);

46

47   /*
48   ** Routine:    ddl_initialize_1
49   **
50   ** Inputs:     DDl_initialize_args *  - args for the restore initialize
                                             call
51   **
52   ** Outputs:    None
53   **
54   ** Return Codes:
55   **    DD_initialize_result *  - result of init function call
56   **
57   ** Purpose:  Function to create a restore session.
58   **
59   ** Intended caller:  Internal Only.
60   **
61   */
```

```
63   DD_initialize_result *
64   ddl_initialize_1_svc(    IN DDl_initialize_args *arg, IN struct svc_req *req )
65   {
66       static DD_initialize_result argzz;

68       InitializeSession(arg, req, &argzz);

70       return &argzz;
71   }
```

```
 73    */
 74    /*************************************************
 75    **
 76    ** Routine:     dd_getservicestatus_l
 77    **
 78    ** Inputs:      dd_getservicestatus_args * - args for the
 79    **                                           getservicestatus call
 80    **
 81    ** Outputs:     None
 82    **
 83    ** Return Codes:
 84    **              dd_getservicestatus_result * - result of status function
 85    **                                             call
 86    **
 87    ** Purpose:     Function to poll for status on a session.
 88    **
 89    ** Intended caller:   Internal Only.
 90    **
 91    *************************************************/
 92    dd_getservicestatus_result *
 93    dd_getservicestatus_l_svc(
 94        dd_getservicestatus_args *arg, IN struct svc_req *req )
 95  1 {
 96  1     static dd_getservicestatus_result argzz;
 97  1
 98  1     GetDispatchStatus(arg, &argzz);
        1
        1     return &argzz;
        1 }
```

```
100    */
101    /*************************************************
102    **
103    ** Routine:     dd_getsessioninfo_l
104    **
105    ** Inputs:      dd_getsessioninfo_args * - args for the getsessioninfo
106    **                                         call
107    **
108    ** Outputs:     None
109    **
110    ** Return Codes:
111    **              SessionBlock * - result of session info call
112    **
113    ** Purpose:     Function to get information on all sessions.
114    **
115    ** Intended caller:   Internal Only.
116    **
117    *************************************************/
118    SessionBlock *
119  1 dd_getsessioninfo_l_svc(
        1     IN dd_getsessioninfo_args *arg, IN struct svc_req *req )
121  1 {
122  1     static SessionBlock argzz;
123  1     static boolean_t first = TRUE;
124  1
125  1     if (first)
126  2     {
127  2         memset(&argzz, 0, sizeof(argzz));
128  1         first = FALSE;
129  1     }
130  2     else
131  2     {
132  2         FreeSessionInfo(&argzz.sess);
        1         argzz.sess = NULL;
        1     }
134  1     GetDispatchInfo(arg, &argzz);
        1
136  1     return &argzz;
137  1 }
```

```
139    /******************************************************
140    **
141    ** Routine:    FreeSessionInfo
142    **
143    ** Inputs:     SessionInfo * - arg to free
144    **
145    ** Outputs:    None
146    **
147    ** Return Codes:
148    **             None
149    **
150    ** Purpose:    Function to free all SessionInfo structures in a list.
151    **
152    ** Intended caller:  Internal Only
153    **
154    *******************************************************/
155    static void FreeSessionInfo(SessionInfo *sess)
156  1 {
157  1     if (sess == NULL)
158  1         return;
159
160  1     if (sess -> next != NULL)
161  1         FreeSessionInfo(sess -> next);
162
163  1     free(sess);
164  1 }
```